Decision Procedures over Sophisticated Fractional Permissions

Le Xuan Bach, Cristian Gherghina, **Aquinas Hobor**
National University of Singapore

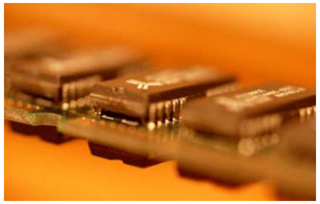
What are Fractional Permissions?

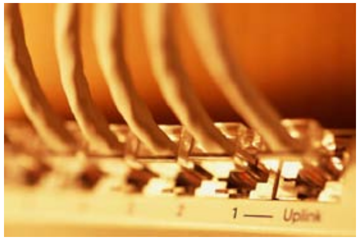
- **Accounting** for **shared control** of a **resource**

What are Fractional Permissions?

- **Accounting** for **shared control** of a **resource**

- Resource: memory cell, file on disk, network connection…

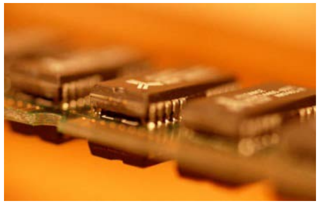


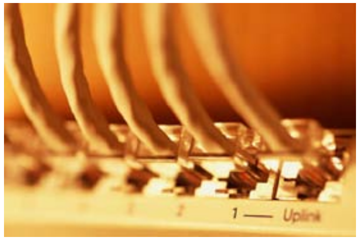


What are Fractional Permissions?

- **Accounting** for **shared control** of a **resource**

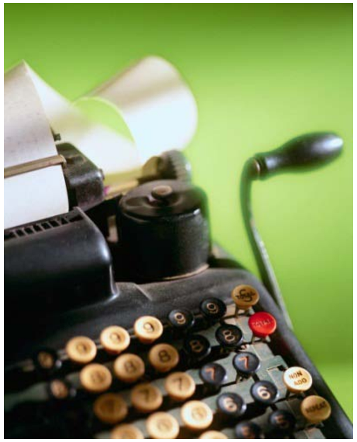
- Resource: memory cell, file on disk, network connection…





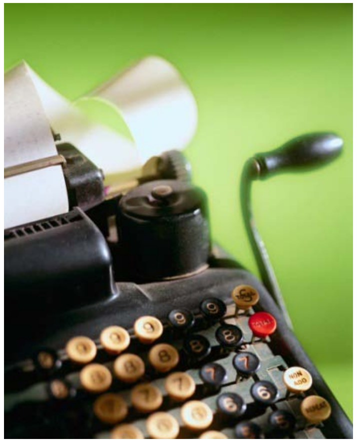
- Shared control: usually between two or more parallel computations

Accounting



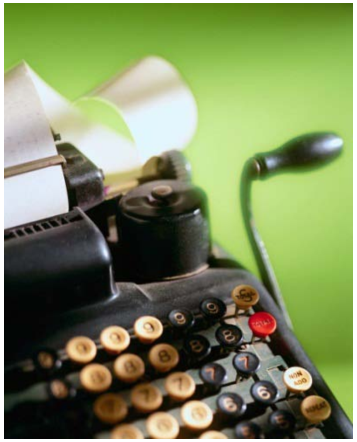
- How we keep track of who owns **how much**
  - e.g., a **share** is a rational in [0, 1]

Accounting



- How we keep track of who owns **how much**
  - e.g., a **share** is a rational in [0, 1]
- And how ownership gets **transferred**
  - we combine shares using partial addition, i.e.
    0.25 + 0.25 = 0.5   but   0.75 + 0.75 is undefined

Accounting



- How we keep track of who owns **how much**
  - e.g., a **share** is a rational in [0, 1]
- And how ownership gets **transferred**
  - we combine shares using partial addition, i.e.
    0.25 + 0.25 = 0.5 but 0.75 + 0.75 is undefined

- Not the same as **policy**, which maps shares to behaviors:
  - {1} : can write to memory cell
  - (0,1] : can read from memory cell
  - {0} : cannot use memory cell

Tree shares

- Rationals do not satisfy exactly the “right” axioms
  - See Parkinson’s thesis or our APLAS 09 paper for why.

[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Tree shares

- Rationals do not satisfy exactly the “right” axioms
  - See Parkinson’s thesis or our APLAS 09 paper for why.

- Solution: use Boolean binary trees for shares
  - Full share: ●

[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

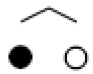
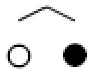
Tree shares

- Rationals do not satisfy exactly the "right" axioms
  - See Parkinson's thesis or our APLAS 09 paper for why.
- Solution: use Boolean binary trees for shares
  - Full share: ●
  - Empty share: ○

[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Tree shares

- Rationals do not satisfy exactly the “right” axioms
  - See Parkinson’s thesis or our APLAS 09 paper for why.

- Solution: use Boolean binary trees for shares
  - Full share: ●
  - Empty share: ○
  - Left half: 
  - Right half: 

[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Tree shares

- Rationals do not satisfy exactly the "right" axioms
  - See Parkinson's thesis or our APLAS 09 paper for why.

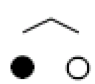
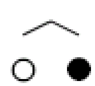
- Solution: use Boolean binary trees for shares
  - Full share: $\bullet$
  - Empty share: $\circ$
  - Left half: $\widehat{\bullet\ \circ}$
  - Right half: $\widehat{\circ\ \bullet}$

These are not the same half share!

[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

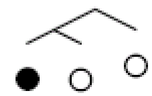
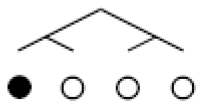
Tree shares

- Rationals do not satisfy exactly the “right” axioms
  - See Parkinson’s thesis or our APLAS 09 paper for why.

- Solution: use Boolean binary trees for shares
  - Full share: ●
  - Empty share: ○
  - Left half: 
  - Right half: 
  - First quarter: 
  - etc.

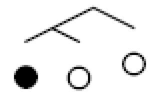
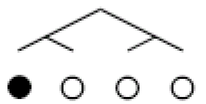
[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Canonical Forms

- Note we wrote the first quarter as  instead of . This is deliberate; the second is not in **canonical form**, which ensures unique representations.

[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Canonical Forms

- Note we wrote the first quarter as  instead of . This is deliberate; the second is not in **canonical form**, which ensures unique representations.

- Define a reflexive, transitive relation $\cong$ from:

[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Canonical Forms

- Note we wrote the first quarter as $\widehat{\widehat{\bullet\ \circ}\ \circ}$ instead of $\widehat{\widehat{\bullet\ \circ}\ \widehat{\circ\ \circ}}$. This is deliberate; the second is not in **canonical form**, which ensures unique representations.

- Define a reflexive, transitive relation $\cong$ from:

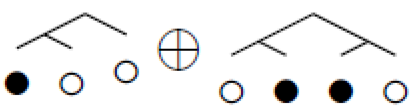
$$\frac{}{\circ \cong \widehat{\circ\ \circ}} \qquad \frac{}{\bullet \cong \widehat{\bullet\ \bullet}}$$

- A tree is in canonical form when it is in the most compact representation under $\cong$.

[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Addition

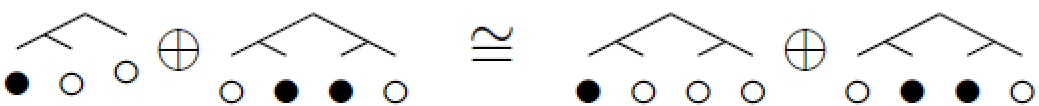
- To add trees (a partial operation), we



[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Addition

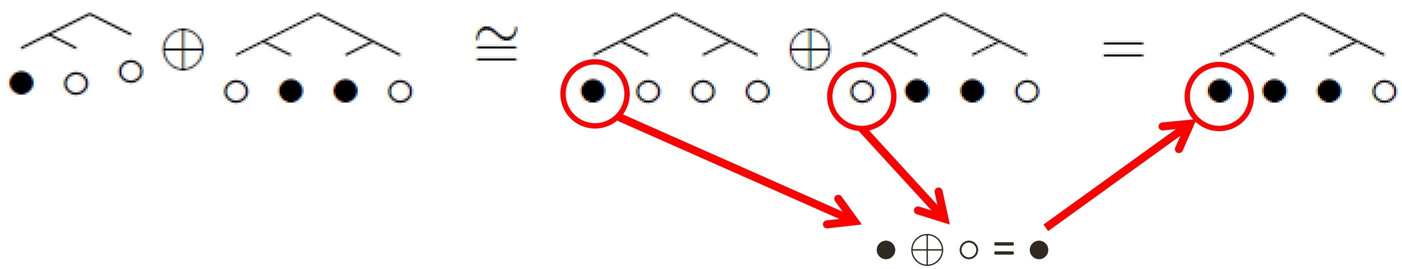
- To add trees (a partial operation), we
  1. Expand them using $\cong$ to the same shape



[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Addition

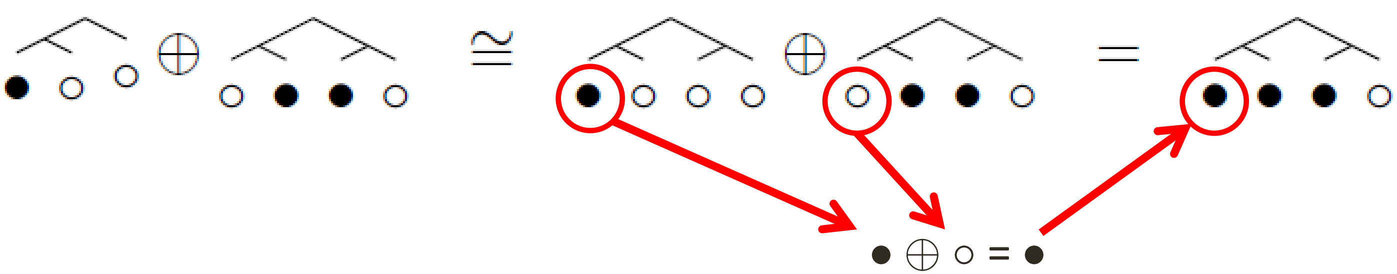
- To add trees (a partial operation), we
  1. Expand them using $\cong$ to the same shape
  2. Join leafwise ($\circ \oplus x = x$ and $x \oplus \circ = x$)



[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Addition

- To add trees (a partial operation), we
  1. Expand them using $\cong$ to the same shape
  2. Join leafwise ($\circ \oplus x = x$ and $x \oplus \circ = x$)

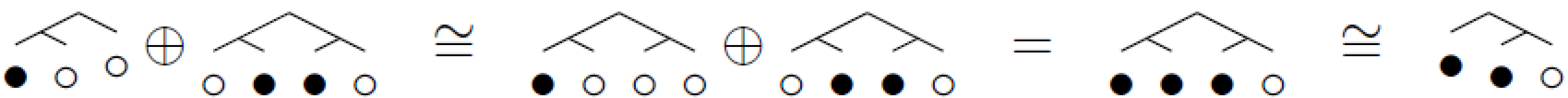


Emphasis: $\bullet \oplus \bullet$ is undefined!

[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Addition

- To add trees (a partial operation), we
  1. Expand them using $\cong$ to the same shape
  2. Join leafwise ($\circ \oplus x = x$ and $x \oplus \circ = x$)
  3. Re-canonicalize



[R. Dockins, A. Hobor, A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting, APLAS 2009]

Using shares in separation logic

- Update “maps-to” to take a tree-share:
  - $e \overset{\pi}{\mapsto} e'$
  - the current heap has a single cell e, whose value is e’, and which is owned with tree-fraction π

- $(5 \mapsto 7) * (8 \mapsto 5) * (5 \mapsto 7)$ =

Using shares in separation logic

- Update “maps-to” to take a tree-share:
  - $e \overset{\pi}{\mapsto} e'$
  - the current heap has a single cell e, whose value is e’, and which is owned with tree-fraction π

- $(5 \mapsto 7) * (8 \mapsto 5) * (5 \mapsto 7) \ = \ \text{False}$

Using shares in separation logic

- Update "maps-to" to take a tree-share:
  - e $\overset{\pi}{\mapsto}$ e'
  - the current heap has a single cell e, whose value is e', and which is owned with tree-fraction π

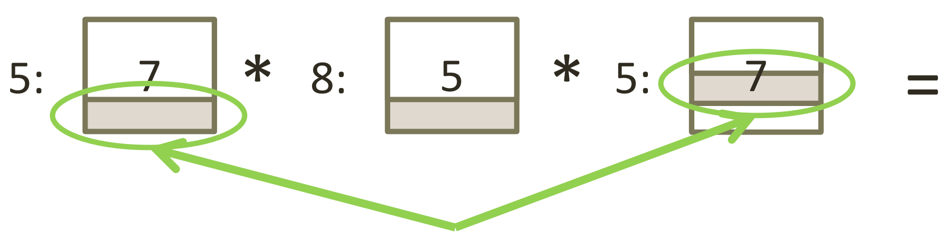
- $(5 \mapsto 7) * (8 \mapsto 5) * (5 \mapsto 7) \;=\;$ False

5: $\boxed{7}$ * 8: $\boxed{5}$ * 5: $\boxed{7}$ = False

These shares cannot be added together!

Using shares in separation logic

- Update “maps-to” to take a tree-share:
  - $e \overset{\pi}{\mapsto} e'$
  - the current heap has a single cell e, whose value is e’, and which is owned with tree-fraction π
- $(5 \mapsto 7) * (8 \mapsto 5) * (5 \mapsto 7) =$

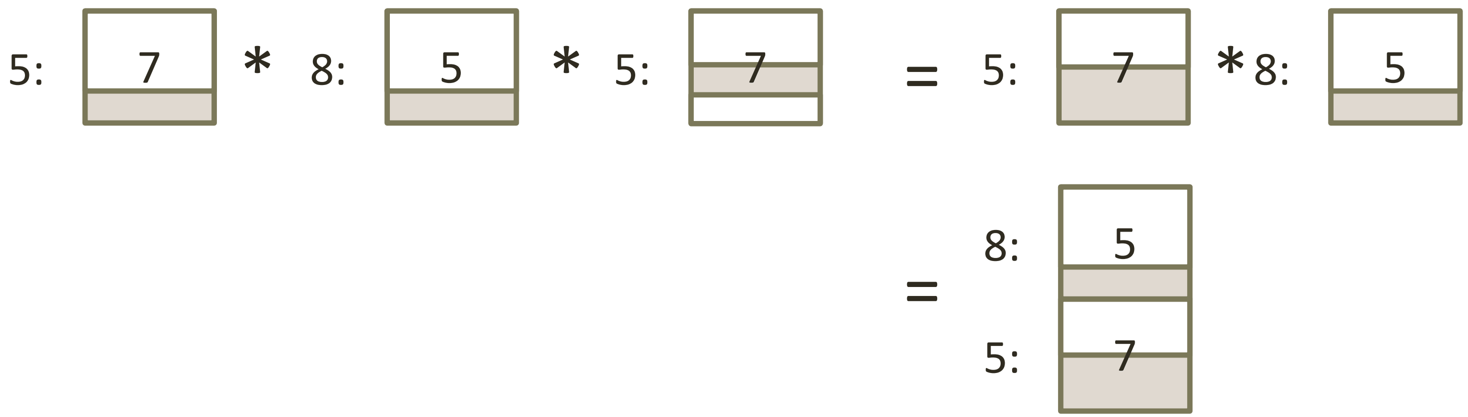


These shares are compatible

Using shares in separation logic

- Update “maps-to” to take a tree-share:
  - $e \overset{\pi}{\mapsto} e'$
  - the current heap has a single cell e, whose value is e’, and which is owned with tree-fraction π

- $(5 \overset{((\bullet\,\circ)\,\circ)}{\mapsto} 7) * (8 \overset{((\bullet\,\circ)\,\circ)}{\mapsto} 5) * (5 \overset{((\circ\,\bullet)\,\circ)}{\mapsto} 7) = (5 \overset{(\bullet\,\circ)}{\mapsto} 7) * (8 \overset{((\bullet\,\circ)\,\circ)}{\mapsto} 5)$



Plan of attack

1. Fractional Shares ✔
2. Verification Tools
3. Our Decision Procedures
4. Completeness
5. Experimental Results

Verification tools

- Once you have a good share model, and have integrated it into a program logic, you would like to use the logic to prove programs.

[H. H. Nguyen, C. David, S. Qin, W. N. Chin. Automated verification of shape and size properties via separation logic. VMCAI 2007]

Verification tools

- Once you have a good share model, and have integrated it into a program logic, you would like to use the logic to prove programs.

- Even better, you'd like to write a program that uses your logic (and thus, the share model) to verify programs for you!

[H. H. Nguyen, C. David, S. Qin, W. N. Chin. Automated verification of shape and size properties via separation logic. VMCAI 2007]

Verification tools

- Once you have a good share model, and have integrated it into a program logic, you would like to use the logic to prove programs.

- Even better, you'd like to write a program that uses your logic (and thus, the share model) to verify programs for you!

- We have modified the HIP/SLEEK toolchain to verify programs using fractional permissions.

[H. H. Nguyen, C. David, S. Qin, W. N. Chin. Automated verification of shape and size properties via separation logic. VMCAI 2007]

Actually, modifying SLEEK is not the major difficulty…

- SLEEK (and many other toolchains) maintains a stable of backend provers for specific domains.
  - Omega (Presburger arithmetic)
  - MONA (bags, etc.)
  - Redlog (real arithmetic)
  - etc.

[A. Hobor, C. Gherghina. Barriers in concurrent separation logic: now with tool support! LMCS Vol. 8, 2011]

Actually, modifying SLEEK is not the major difficulty…

- SLEEK (and many other toolchains) maintains a stable of backend provers for specific domains.
  - Omega (Presburger arithmetic)
  - MONA (bags, etc.)
  - Redlog (real arithmetic)
  - etc.

- We fit into this pattern: **our major accomplishment is a backend prover for tree-shares. Our prover should be re-usable (as a library or standalone) in many other toolchains.**

[A. Hobor, C. Gherghina. Barriers in concurrent separation logic: now with tool support! LMCS Vol. 8, 2011]

SLEEK's job

- Accordingly, SLEEK's job is to isolate the "share-related" subproblems from SL entailments.

[A. Hobor, C. Gherghina. Barriers in concurrent separation logic: now with tool support! LMCS Vol. 8, 2011]

SLEEK's job

- Accordingly, SLEEK's job is to isolate the "share-related" subproblems from SL entailments.

- Really simple example:
  - from $x \overset{s_1}{\mapsto} v \wedge s_1 = \widehat{\bullet\ \circ} \wedge s_2 = \widehat{\circ\ \bullet} \quad \vdash \quad x \overset{s_2}{\mapsto} v$

[A. Hobor, C. Gherghina. Barriers in concurrent separation logic: now with tool support! LMCS Vol. 8, 2011]

SLEEK's job

- Accordingly, SLEEK's job is to isolate the "share-related" subproblems from SL entailments.

- Really simple example:

  - from $x \overset{s_1}{\mapsto} v \wedge s_1 = \widehat{\bullet\ \circ} \wedge s_2 = \widehat{\circ\ \bullet} \vdash x \overset{s_2}{\mapsto} v$

  - we reach $s_1 = \widehat{\bullet\ \circ} \wedge s_2 = \widehat{\circ\ \bullet} \vdash s_1 = s_2,$

[A. Hobor, C. Gherghina. Barriers in concurrent separation logic: now with tool support! LMCS Vol. 8, 2011]

SLEEK's job

- Accordingly, SLEEK's job is to isolate the "share-related" subproblems from SL entailments.

- Really simple example:

  - from $x \overset{s_1}{\mapsto} v \wedge s_1 = \widehat{\bullet\ \circ} \wedge s_2 = \widehat{\circ\ \bullet} \quad \vdash \quad x \overset{s_2}{\mapsto} v$

  - we reach $s_1 = \widehat{\bullet\ \circ} \wedge s_2 = \widehat{\circ\ \bullet} \vdash s_1 = s_2$,

  - which is satisfied by a decidable equality check: $\widehat{\bullet\ \circ} \overset{?}{=} \widehat{\circ\ \bullet}$ (false).

[A. Hobor, C. Gherghina. Barriers in concurrent separation logic: now with tool support! LMCS Vol. 8, 2011]

Formal statement of problem

- SLEEK outputs systems of share equations:
  - $\phi \equiv \exists v.\ \phi$
    $\mid\ \phi_1 \wedge \phi_2$
    $\mid\ v_1 \oplus v_2 = v_3$
    $\mid\ v_1 = v_2$
    $\mid\ v = \chi$

Formal statement of problem

- SLEEK outputs systems of share equations:
  - $\phi \equiv \exists \text{v}.\ \phi$

    $\mid \ \phi_1 \wedge \phi_2$

    $\mid \ \text{v}_1 \oplus \text{v}_2 = \text{v}_3$

    $\mid \ \text{v}_1 = \text{v}_2$

    $\mid \ \text{v} = \chi$

These are the share constants, like $\widehat{\bullet\ \circ}$

Formal statement of problem

- SLEEK outputs systems of share equations:
  - $\phi \equiv \exists \mathrm{v}.\ \phi$
    $\mid\ \phi_1 \wedge \phi_2$
    $\mid\ \mathrm{v}_1 \oplus \mathrm{v}_2 = \mathrm{v}_3$
    $\mid\ \mathrm{v}_1 = \mathrm{v}_2$
    $\mid\ \mathrm{v} = \chi$

SLEEK does not need to know much about the underlying domain of tree-shares to isolate the associated facts

Formal statement of problem

- SLEEK outputs systems of share equations:
  - $\phi \equiv \exists v.\ \phi$
    $\mid \phi_1 \wedge \phi_2$
    $\mid v_1 \oplus v_2 = v_3$
    $\mid v_1 = v_2$
    $\mid v = \chi$

This output format is a useful modularity boundary we discovered by experimentation

Formal statement of problem

- SLEEK outputs systems of share equations:
  - $$\begin{aligned}\phi \equiv\ & \exists\, \mathrm{v}.\ \phi \\ \mid\ & \phi_1 \wedge \phi_2 \\ \mid\ & \mathrm{v}_1 \oplus \mathrm{v}_2 = \mathrm{v}_3 \\ \mid\ & \mathrm{v}_1 = \mathrm{v}_2 \\ \mid\ & \mathrm{v} = \chi\end{aligned}$$

- SLEEK can then ask two kinds of questions:

Formal statement of problem

- SLEEK outputs systems of share equations:
  - $\phi \equiv \exists \text{v}.\ \phi$
    $\quad | \ \phi_1 \wedge \phi_2$
    $\quad | \ \text{v}_1 \oplus \text{v}_2 = \text{v}_3$
    $\quad | \ \text{v}_1 = \text{v}_2$
    $\quad | \ \text{v} = \chi$

- SLEEK can then ask two kinds of questions:
  - (SAT) Is a given system satisfiable? (Used to prune unfeasible verification paths)

Formal statement of problem

- SLEEK outputs systems of share equations:
  - $\phi \equiv \exists v.\ \phi$
    $\mid\ \phi_1 \wedge \phi_2$
    $\mid\ v_1 \oplus v_2 = v_3$
    $\mid\ v_1 = v_2$
    $\mid\ v = \chi$

- SLEEK can then ask two kinds of questions:
  - (SAT) Is a given system satisfiable? (Used to prune unfeasible verification paths)
  - (IMPL) Does one system of equations imply another?

Plan of attack

1. Fractional Shares ✔
2. Verification Tools ✔
3. Our Decision Procedures
4. Completeness
5. Experimental Results

Why the problem is hard

- Like the rationals, the space of tree-shares is **dense**: that is, given any nonempty share, you can divide it into two nonempty shares
  - Need this to verify divide-and-conquer algorithms!

Why the problem is hard

- Like the rationals, the space of tree-shares is **dense**: that is, given any nonempty share, you can divide it into two nonempty shares
  - Need this to verify divide-and-conquer algorithms!

- Thus, it appears as though finite search is not enough: there could always be a solution to SAT (or a counterexample to IMPL) “just a little deeper”

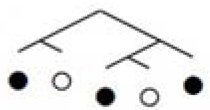
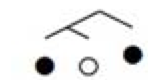
Why the problem is hard

- Like the rationals, the space of tree-shares is **dense**: that is, given any nonempty share, you can divide it into two nonempty shares
  - Need this to verify divide-and-conquer algorithms!

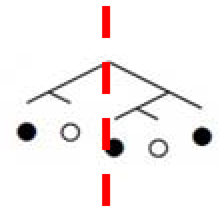
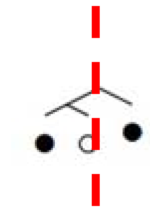
- Thus, it appears as though finite search is not enough: there could always be a solution to SAT (or a counterexample to IMPL) “just a little deeper”

- Surprisingly, this intution is wrong: we do a **shape-guided finite search**, armed with some **completeness results** that say our finite search is sufficient.

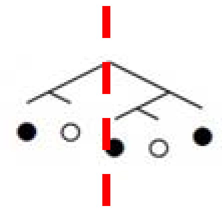
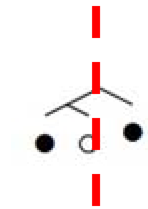
Decomposition (SAT)

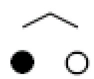
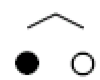
- We want to know if the following system is satisfiable:
  - x $\oplus$  = y $\wedge$ y $\oplus$ z = 

Decomposition (SAT)

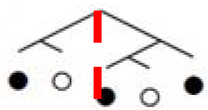
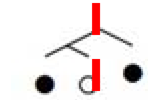
- We want to know if the following system is satisfiable:
  - $x \oplus$  $= y \quad \wedge \quad y \oplus z =$ 
- We split into **two** systems…

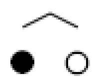
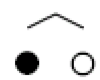
Decomposition (SAT)

- We want to know if the following system is satisfiable:
  - $x \oplus$  $= y \quad \wedge \quad y \oplus z =$ 
- We split into **two** systems…

1. $x_l \oplus$  $= y_l \quad \wedge \quad y_l \oplus z_l =$ 
2. $x_r \oplus$  $= y_r \quad \wedge \quad y_r \oplus z_r = \bullet$

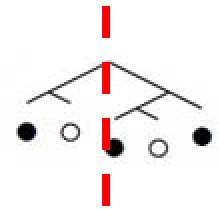
Decomposition (SAT)

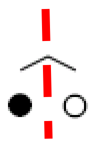
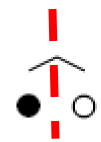
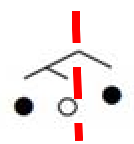
- We want to know if the following system is satisfiable:
  - $x \oplus$  $= y \quad \wedge \quad y \oplus z =$ 
- We split into **two** systems…

1. $x_l \oplus$  $= y_l \quad \wedge \quad y_l \oplus z_l =$ 
2. $x_r \oplus$  $= y_r \quad \wedge \quad y_r \oplus z_r = \bullet$

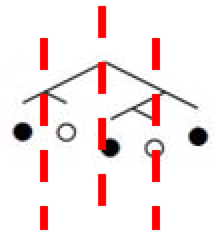
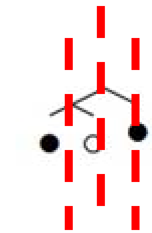
Theorem: The original system is satisfiable if and only if both subsystems are satisfiable

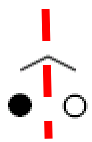
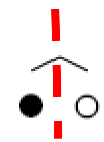
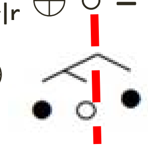
Decomposition (SAT)

- We want to know if the following system is satisfiable:
  - $x \oplus$  $= y \quad \wedge \quad y \oplus z =$ 
- We split into **two** systems… and then keep splitting…

1. $x_l \oplus$  $= y_l \quad \wedge \quad y_l \oplus z_l =$ 
2. $x_r \oplus$  $= y_r \quad \wedge \quad y_r \oplus z_r =$ 

Decomposition (SAT)

- We want to know if the following system is satisfiable:
  - $x \oplus$  $= y \quad \wedge \quad y \oplus z =$ 
- We split into **two** systems… and then keep splitting…

1. $x_l \oplus$  $= y_l \quad \wedge \quad y_l \oplus z_l =$ 
   a) $x_{ll} \oplus \bullet = y_{ll} \quad \wedge \quad y_{ll} \oplus z_{ll} = \bullet$
   b) $x_{lr} \oplus \circ = y_{lr} \quad \wedge \quad y_{lr} \oplus z_{lr} = \circ$
2. $x_r \oplus$  $= y_r \quad \wedge \quad y_r \oplus z_r =$ 
   a) $x_{rl} \oplus$  $= y_{rl} \quad \wedge \quad y_{rl} \oplus z_{rl} = \bullet$
   b) $x_{rr} \oplus \bullet = y_{rr} \quad \wedge \quad y_{rr} \oplus z_{rr} = \bullet$

Once every constant is ● or ○…

- We apply a completeness theorem (shown later) that tells us that **if there is a solution at all, there must exist a solution at the height of the system**

Once every constant is ● or ○…

- We apply a completeness theorem (shown later) that tells us that **if there is a solution at all, there must exist a solution at the height of the system**

- That lets us translate our problem over shares into a Boolean SAT (with existentials) problem

Once every constant is ● or ○…

- We apply a completeness theorem (shown later) that tells us that **if there is a solution at all, there must exist a solution at the height of the system**

- That lets us translate our problem over shares into a Boolean SAT (with existentials) problem
  - Example: $x \oplus y = \bullet \quad \rightsquigarrow \quad (x \vee y) \wedge (\neg x \vee \neg y)$

Once every constant is $\bullet$ or $\circ$…

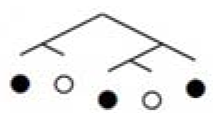
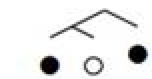
- We apply a completeness theorem (shown later) that tells us that **if there is a solution at all, there must exist a solution at the height of the system**

- That lets us translate our problem over shares into a Boolean SAT (with existentials) problem
  - Example: $x \oplus y = \bullet \quad \rightsquigarrow \quad (x \vee y) \wedge (\neg x \vee \neg y)$

- Then we hand this problem to an SMT solver (e.g. Z3)

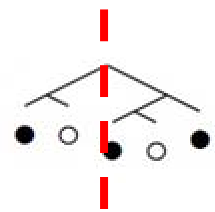
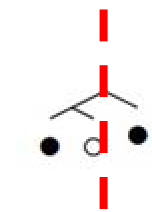
Decomposition (IMPL)

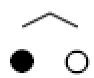
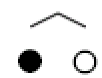
- The procedure for IMPL is very similar, but we have to decompose across the entailment:

  - From $x \oplus$  $= y \quad \vdash \quad \exists z.\ y \oplus z =$ 

Decomposition (IMPL)

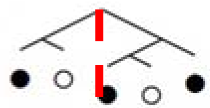
- The procedure for IMPL is very similar, but we have to decompose across the entailment:

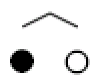
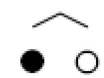
  - From $x \oplus$  $= y \quad \vdash \quad \exists z.\ y \oplus z =$ 

    1. $x_l \oplus$  $= y_l \quad \vdash \quad \exists z_l.\ y_l \oplus z_l =$ 
    2. $x_r \oplus$  $= y_r \quad \vdash \quad \exists z_r.\ y_r \oplus z_r = \bullet$

Decomposition (IMPL)

- The procedure for IMPL is very similar, but we have to decompose across the entailment:

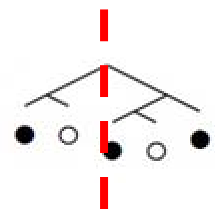
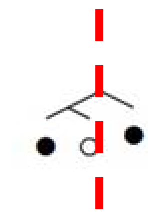
  - From $x \oplus$  $= y \quad \vdash \quad \exists z.\ y \oplus z =$ 

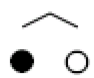
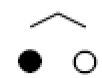
    1. $x_l \oplus$  $= y_l \quad \vdash \quad \exists z_l.\ y_l \oplus z_l =$ 
    2. $x_r \oplus$  $= y_r \quad \vdash \quad \exists z_r.\ y_r \oplus z_r = \bullet$

Theorem: The original entailment holds if and only if both subentailments hold

Decomposition (IMPL)

- The procedure for IMPL is very similar, but we have to decompose across the entailment:

  - From $x \oplus$  $= y \quad \vdash \quad \exists z.\ y \oplus z =$ 

    1. $x_l \oplus$  $= y_l \quad \vdash \quad \exists z_l.\ y_l \oplus z_l =$ 
    2. $x_r \oplus$  $= y_r \quad \vdash \quad \exists z_r.\ y_r \oplus z_r = \bullet$

- Once we have reached height zero, we apply a more complicated completeness theorem and then again translate to Boolean SAT for Z3.

Plan of attack

1. Fractional Shares ✔
2. Verification Tools ✔
3. Our Decision Procedures ✔
4. Completeness
5. Experimental Results

Completeness theorem (SAT)

- Theorem 1: finite search for SAT
  - Given Σ, $\exists\sigma.\ \sigma \vDash \Sigma$ iff $\exists\sigma.\ |\sigma| = |\Sigma| \wedge \sigma \vDash \Sigma$

Completeness theorem (SAT)

- Theorem 1: finite search for SAT
  - Given Σ, $\exists\sigma.\ \sigma \models \Sigma$ iff $\exists\sigma.\ |\sigma| = |\Sigma| \wedge \sigma \models \Sigma$

A system of equations

Completeness theorem (SAT)

- Theorem 1: finite search for SAT
  - Given Σ, $\exists\sigma.\ \sigma \vDash \Sigma$ iff $\exists\sigma.\ |\sigma| = |\Sigma| \wedge \sigma \vDash \Sigma$

A system of equations

A solution (map from tree variables to tree constants)

Completeness theorem (SAT)

- Theorem 1: finite search for SAT
  - Given Σ, $\exists\sigma.\ \sigma \models \Sigma$ iff $\exists\sigma.\ |\sigma| = |\Sigma| \wedge \sigma \models \Sigma$

A system of equations

A solution (map from tree variables to tree constants)

Satisfaction: when variables in Σ are assigned values from σ, then every equation holds.

Completeness theorem (SAT)

- Theorem 1: finite search for SAT
  - Given Σ, $\exists \sigma. \sigma \models \Sigma$ iff $\exists \sigma. |\sigma| = |\Sigma| \wedge \sigma \models \Sigma$

A system of equations

Height: highest tree-constant contained in σ or Σ.

A solution (map from tree variables to tree constants)

Satisfaction: when variables in Σ are assigned values from σ, then every equation holds.

Completeness theorem (SAT)

- Theorem 1: finite search for SAT
  - Given Σ, $\exists\sigma.\ \sigma \models \Sigma$ iff $\exists\sigma.\ |\sigma|=|\Sigma| \wedge \sigma \models \Sigma$

- Strategy:

Completeness theorem (SAT)

- Theorem 1: finite search for SAT
  - Given Σ, $\exists\sigma.\ \sigma \models \Sigma$ iff $\exists\sigma.\ |\sigma| = |\Sigma| \ \wedge\ \sigma \models \Sigma$

- Strategy:
  - Definition by example: rounding a tree

Completeness theorem (SAT)

- Theorem 1: finite search for SAT
  - Given Σ, $\exists\sigma.\ \sigma \vDash \Sigma$ iff $\exists\sigma.\ |\sigma|=|\Sigma| \ \wedge\ \sigma \vDash \Sigma$

- Strategy:
  - Definition by example: rounding a tree

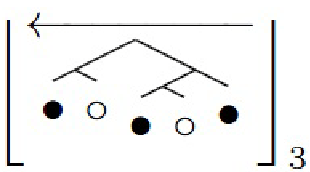
  - Proofs by example: properties of rounding

Completeness theorem (SAT)

- Theorem 1: finite search for SAT
  - Given Σ, $\exists\sigma.\ \sigma \models \Sigma$ iff $\exists\sigma.\ |\sigma| = |\Sigma| \ \land\ \sigma \models \Sigma$
- Strategy:
  - Definition by example: rounding a tree
  - Proofs by example: properties of rounding
  - Proof sketch of main theorem

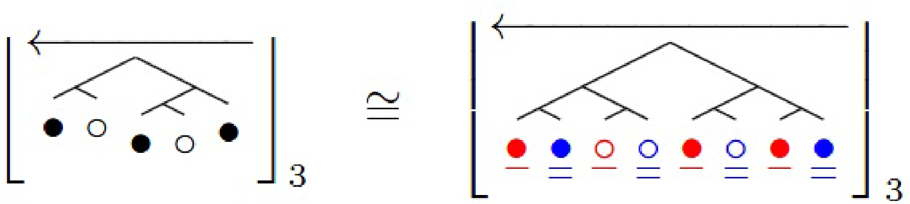
Tree rounding

- Define $\lfloor \overleftarrow{\tau} \rfloor_n$ “left round tree τ to height n” as follows:



Tree rounding

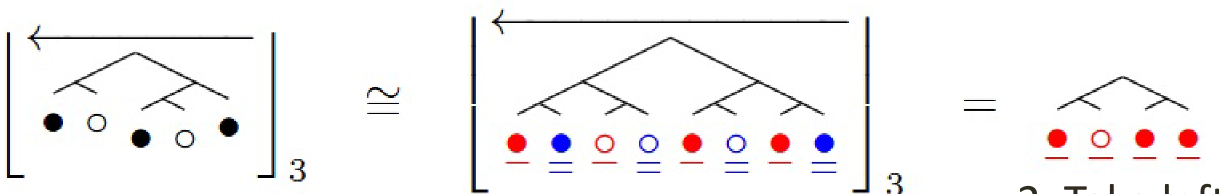
- Define $\lfloor \overleftarrow{\tau} \rfloor_n$ "left round tree τ to height n" as follows:
  1. Unfold τ to height n (height starts at 0)



1. Unfold to height 3

Tree rounding

- Define $\lfloor \overleftarrow{\tau} \rfloor_n$ "left round tree τ to height n" as follows:
  1. Unfold τ to height n (height starts at 0)
  2. Take every **left** leaf at height n

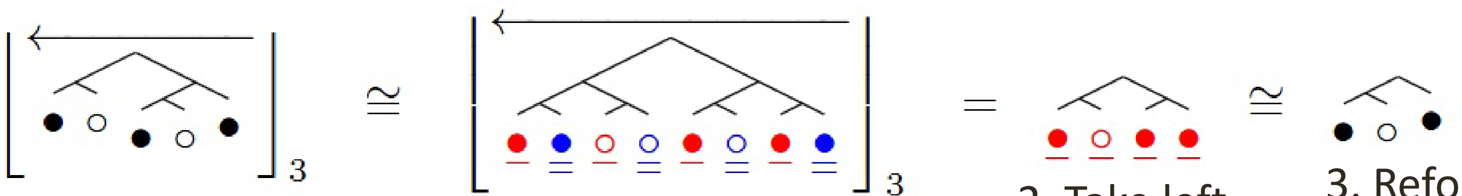


1. Unfold to height 3

2. Take left leaves

Tree rounding

- Define $\lfloor \overleftarrow{\tau} \rfloor_n$ "left round tree τ to height n" as follows:
  1. Unfold τ to height n (height starts at 0)
  2. Take every **left** leaf at height n
  3. Refold as needed



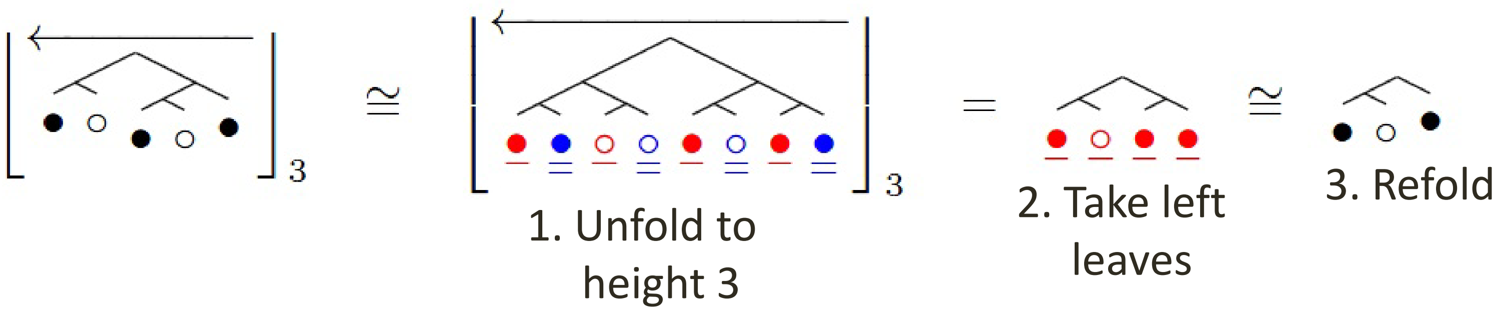
1. Unfold to height 3

2. Take left leaves

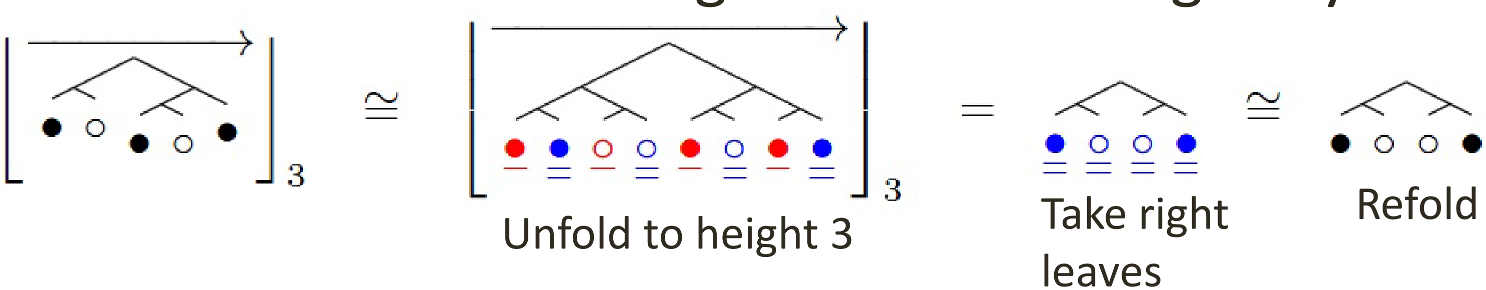
3. Refold

Tree rounding

- Define $\lfloor \overleftarrow{\tau} \rfloor_n$ “left round tree τ to height n” as follows:
  1. Unfold τ to height n (height starts at 0)
  2. Take every **left** leaf at height n
  3. Refold as needed



- We can also define “right round” analogously:



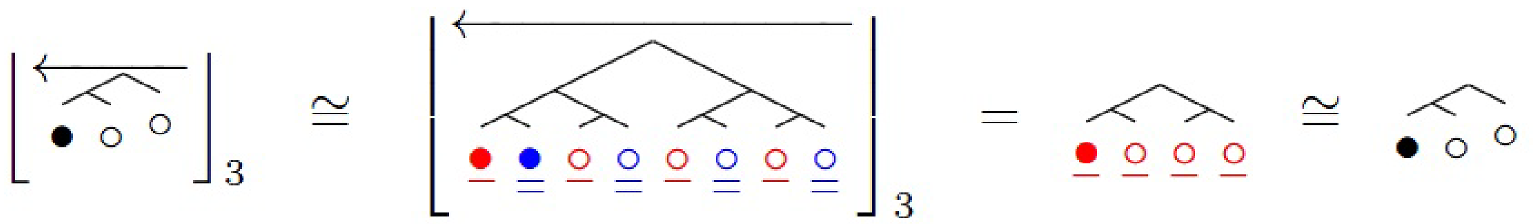
Key properties of rounding

1. Rounding a tree of height n to any height **strictly greater** than n does not change the tree.

Key properties of rounding

1. Rounding a tree of height n to any height **strictly greater** than n does not change the tree.

- “Proof.”



Unfold

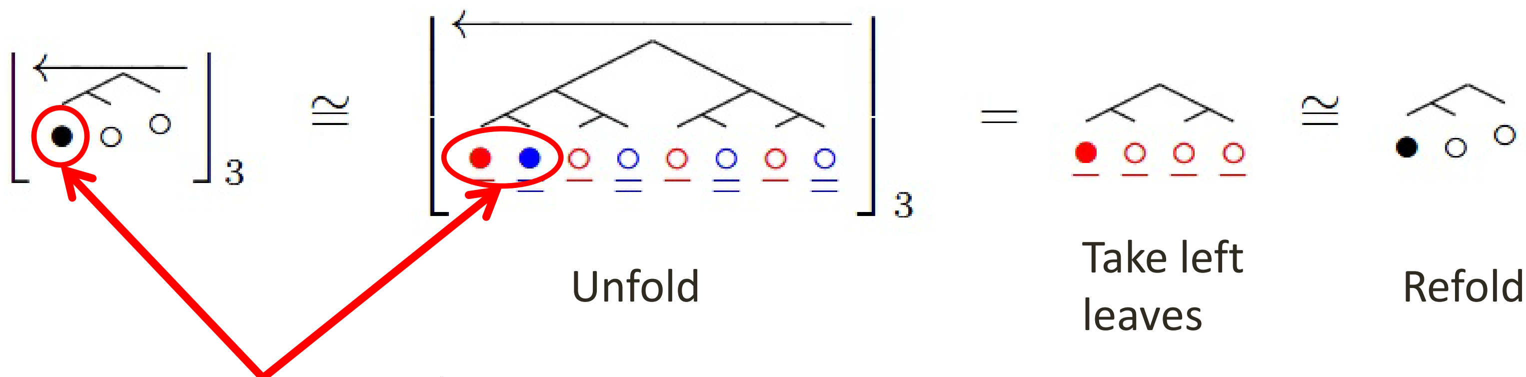
Take left leaves

Refold

Key properties of rounding

1. Rounding a tree of height n to any height **strictly greater** than n does not change the tree.

- "Proof."

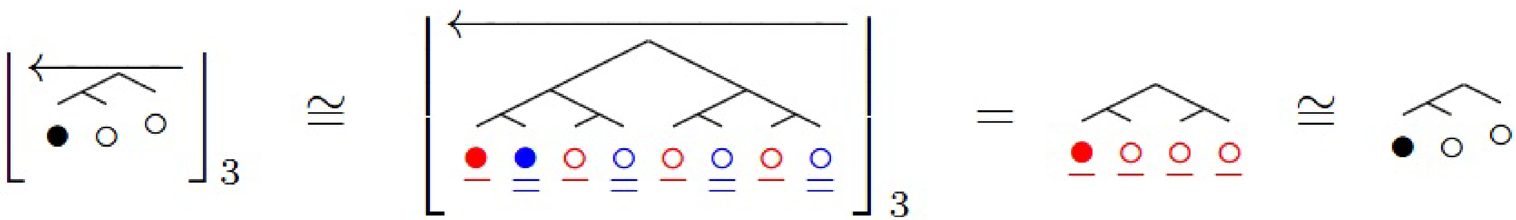


In general, $x \rightsquigarrow x \frown x$, and then we take the left/right, leaving x

Key properties of rounding

1. Rounding a tree of height n to any height **strictly greater** than n does not change the tree.

- "Proof."



Unfold

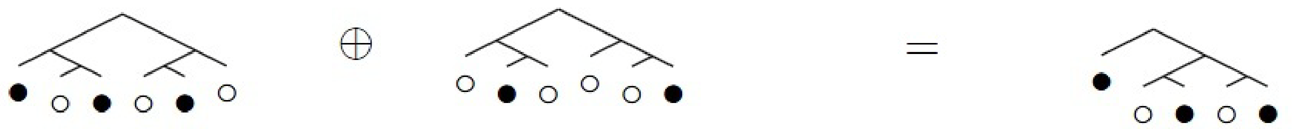
Take left leaves

Refold

2. If $\tau_1 \oplus \tau_2 = \tau_3$, then $\lfloor \overleftarrow{\tau_1} \rfloor_n \oplus \lfloor \overleftarrow{\tau_2} \rfloor_n = \lfloor \overleftarrow{\tau_3} \rfloor_n$.

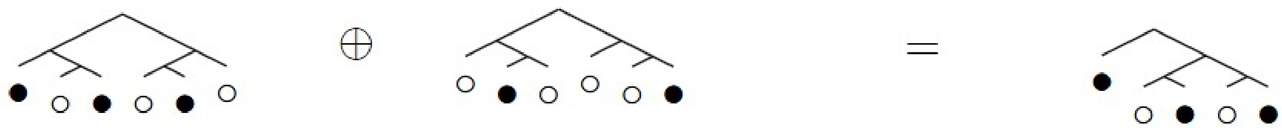
“Proof.”

Premise

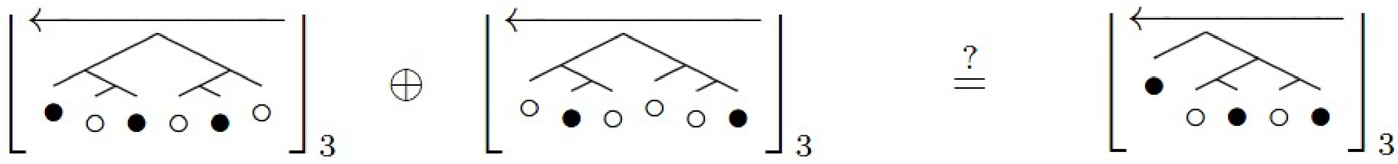


“Proof.”

Premise



Apply round

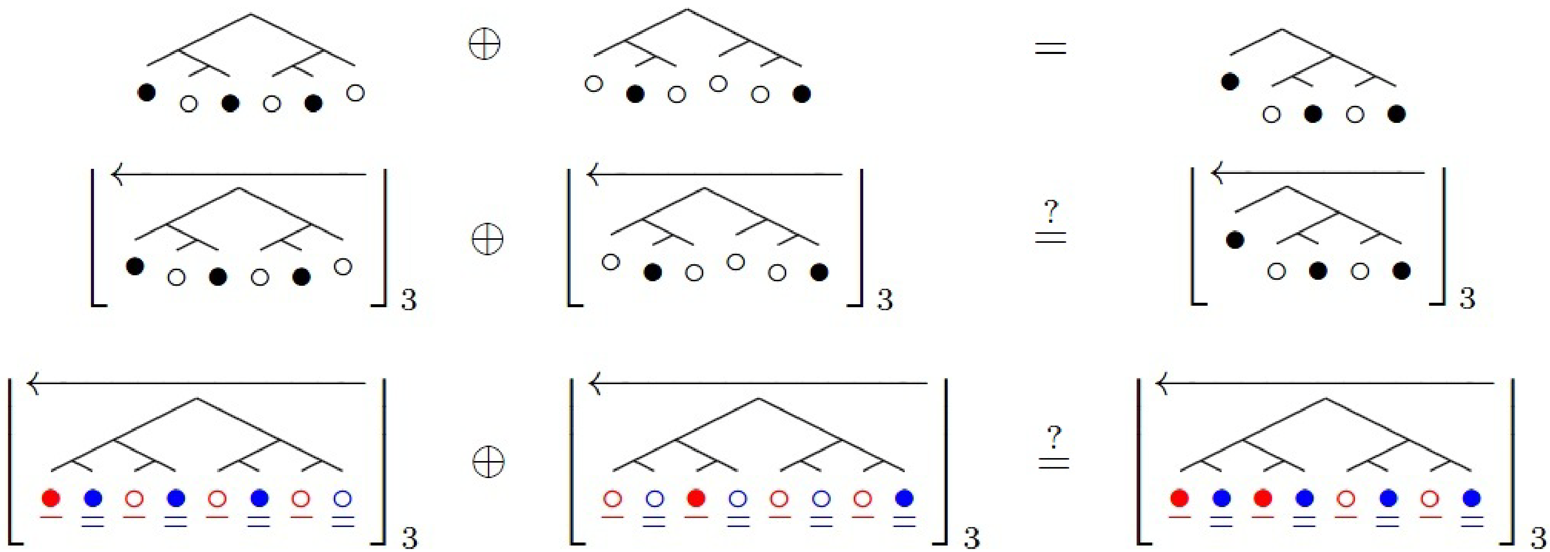


“Proof.”

Premise

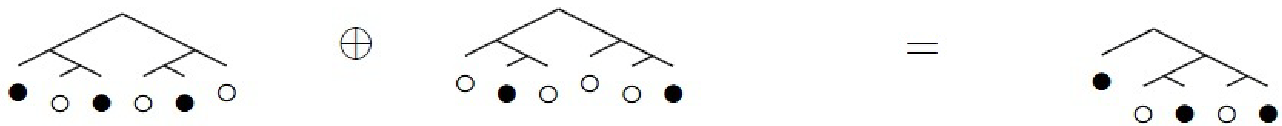
Apply round

Unfold

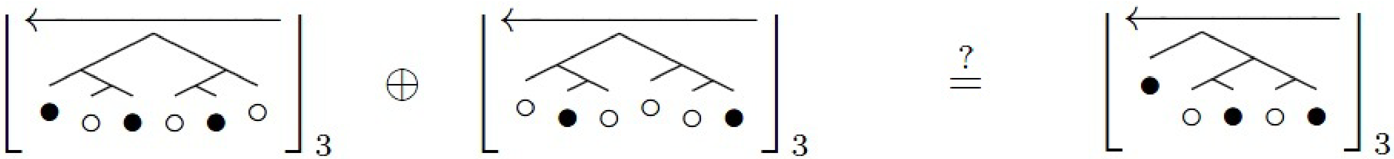


“Proof.”

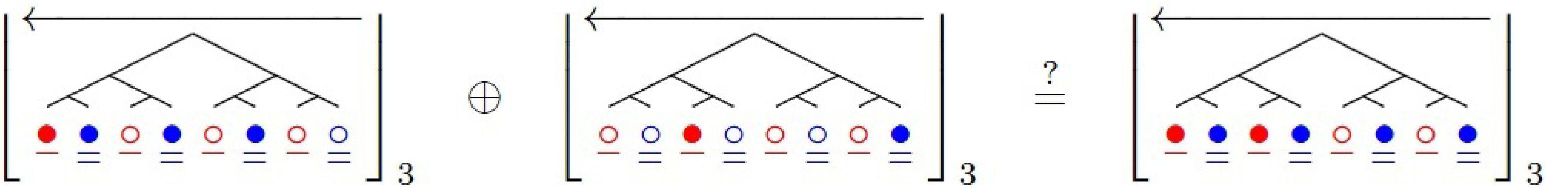
Premise



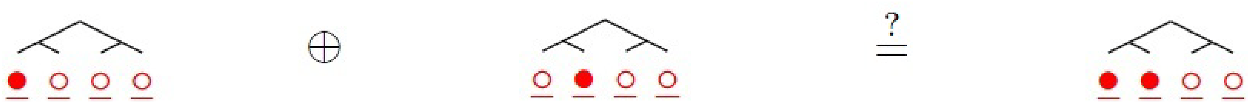
Apply round



Unfold



Take left leaves



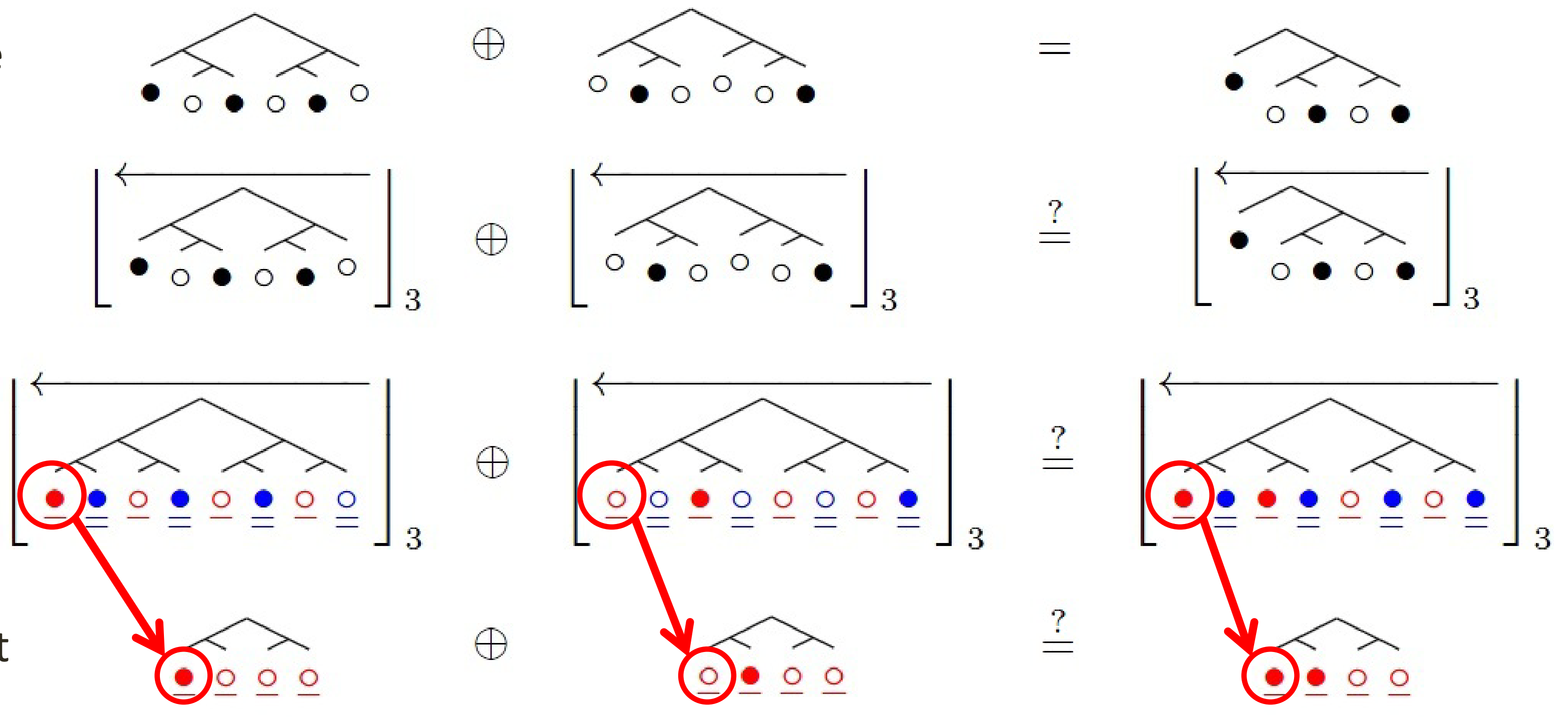
“Proof.”

Premise

Apply round

Unfold

Take left leaves



They joined before – joining occurs leafwise – so of course they join after!

“Proof.”

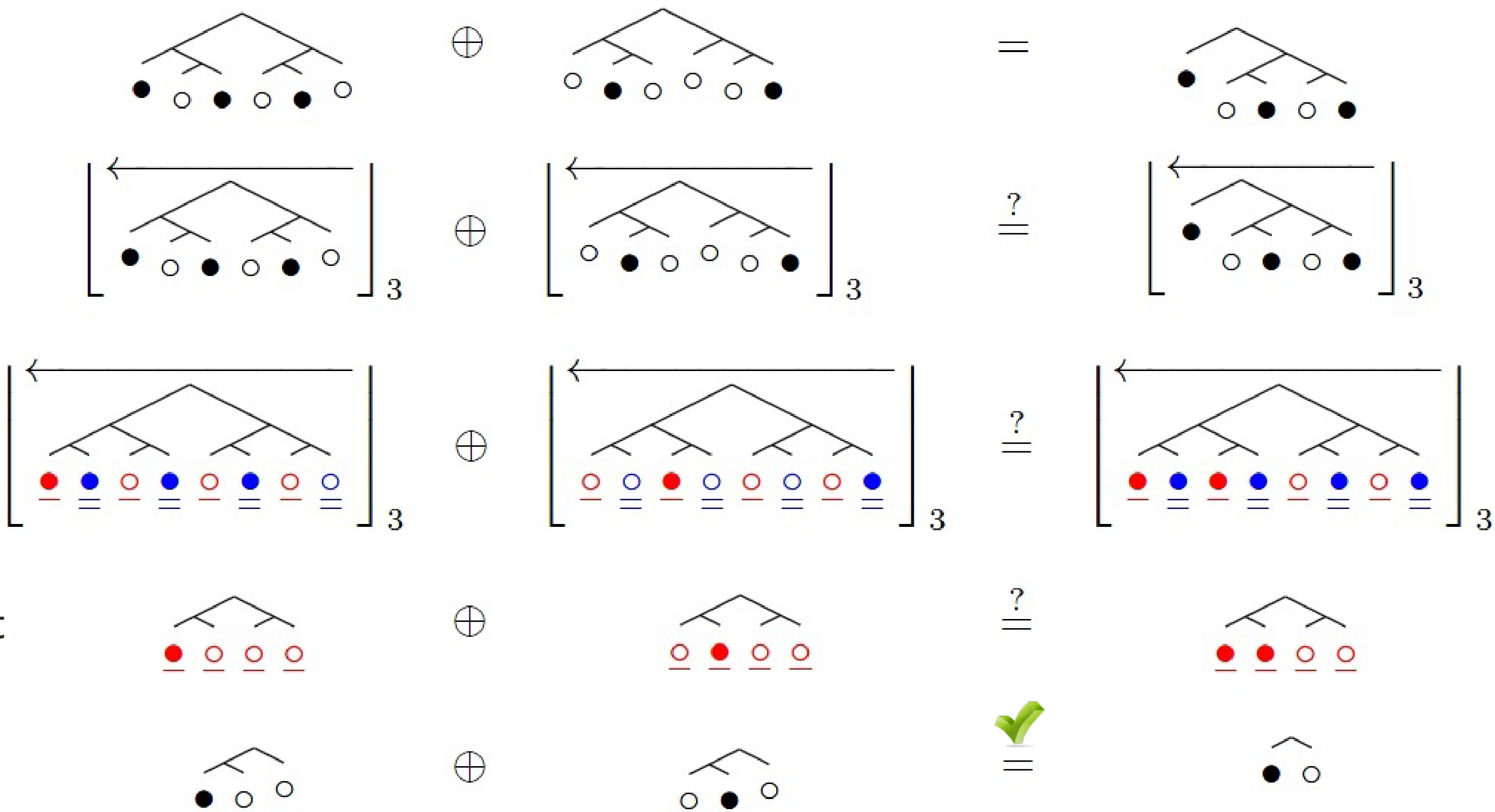
Premise

Apply round

Unfold

Take left leaves

Refold



Proof sketch: finite SAT

- Theorem 1: finite search for SAT
  - Given Σ, $\exists\sigma.\ \sigma \vDash \Sigma$ iff $\exists\sigma.\ |\sigma|=|\Sigma| \wedge \sigma \vDash \Sigma$

- $\leftarrow$: trivial.

- $\rightarrow$: Take σ and repeatedly round it until it is of height $|\Sigma|$. Each equation in $|\Sigma|$ will still hold as long as we also round all constants (property 2), and since we are never rounding to height $|\Sigma|$, the constants in Σ are not changing (property 1), i.e., it is the same system of equations.

Completeness theorem (IMPL)

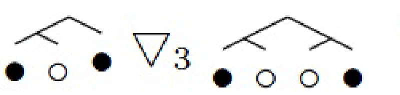
- Theorem 2: finite search for IMPL
  - Given Σ and Σ', $(\forall \sigma.\ \sigma \models \Sigma \rightarrow \sigma \models \Sigma')$ iff
    $(\forall \sigma.\ |\sigma| = |\Sigma| \rightarrow \sigma \models \Sigma \rightarrow \sigma \models \Sigma')$

Completeness theorem (IMPL)

- Theorem 2: finite search for IMPL
  - Given Σ and Σ', $(\forall\,\sigma.\ \sigma \vDash \Sigma\ \rightarrow \sigma \vDash \Sigma')$ iff
    $(\forall\,\sigma.\ |\sigma|=|\Sigma| \rightarrow \sigma \vDash \Sigma\ \rightarrow \sigma \vDash \Sigma')$
- Strategy:
  - Definition by example: averaging two trees
  - Proofs by example: properties of averaging
  - Proof sketch of main theorem

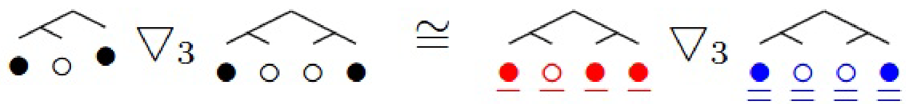
Averaging Trees

- Define $\tau_l \bigtriangledown_n \tau_r$ “averaging two trees at height n”:



Averaging Trees

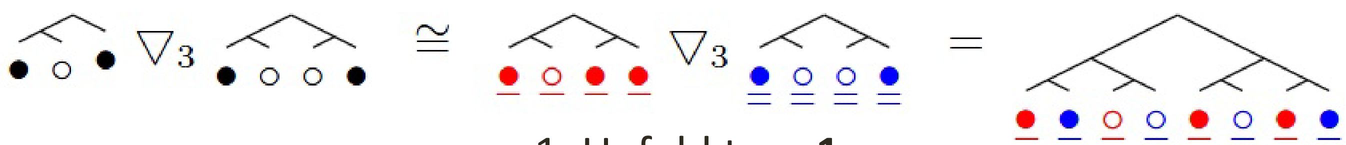
- Define $\tau_l \ \nabla_n \ \tau_r$ "averaging two trees at height n":
  1. Unfold τ to height ***n*-1**



1. Unfold to **n-1**

Averaging Trees

- Define $\tau_l \bigtriangledown_n \tau_r$ "averaging two trees at height n":
  1. Unfold τ to height ***n*-1**
  2. Combine pairwise: left argument become left leaves in result; right argument become right leaves

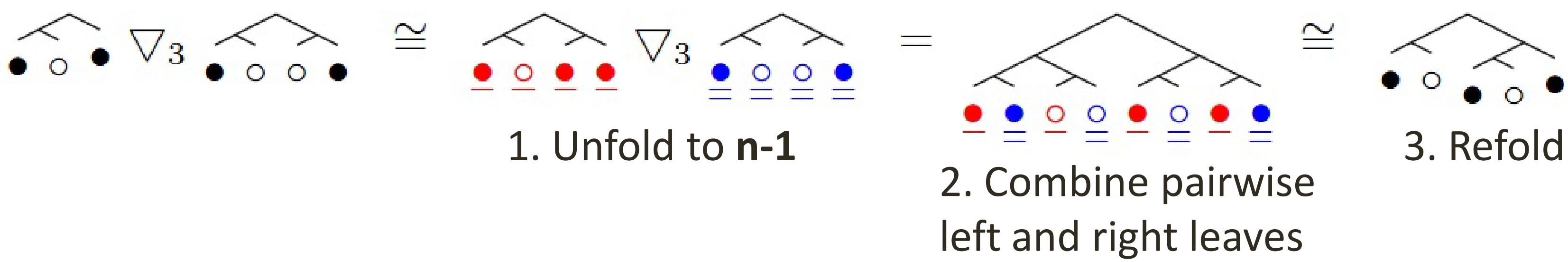


1. Unfold to **n-1**

2. Combine pairwise left and right leaves

Averaging Trees

- Define $\tau_l \ \nabla_n \ \tau_r$ "averaging two trees at height n":
  1. Unfold τ to height ***n*-1**
  2. Combine pairwise: left argument become left leaves in result; right argument become right leaves
  3. Refold as needed



Key properties of averaging

1. Averaging is the inverse of rounding, i.e.,

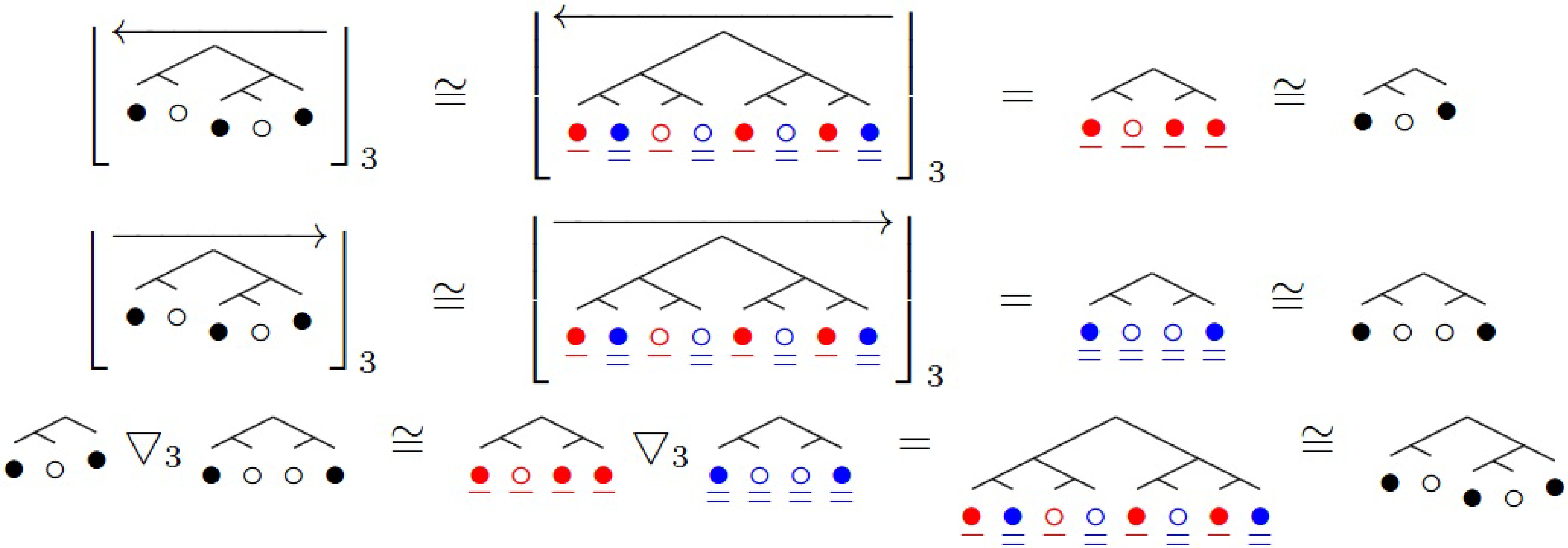
$$\lfloor \overleftarrow{\tau} \rfloor_n \ \nabla_n \ \lfloor \overrightarrow{\tau} \rfloor_n = \tau$$

Key properties of averaging

1. Averaging is the inverse of rounding, i.e.,

$$\lfloor \overleftarrow{\tau} \rfloor_n \ \nabla_n \ \lfloor \overrightarrow{\tau} \rfloor_n = \tau$$

- “Proof.”

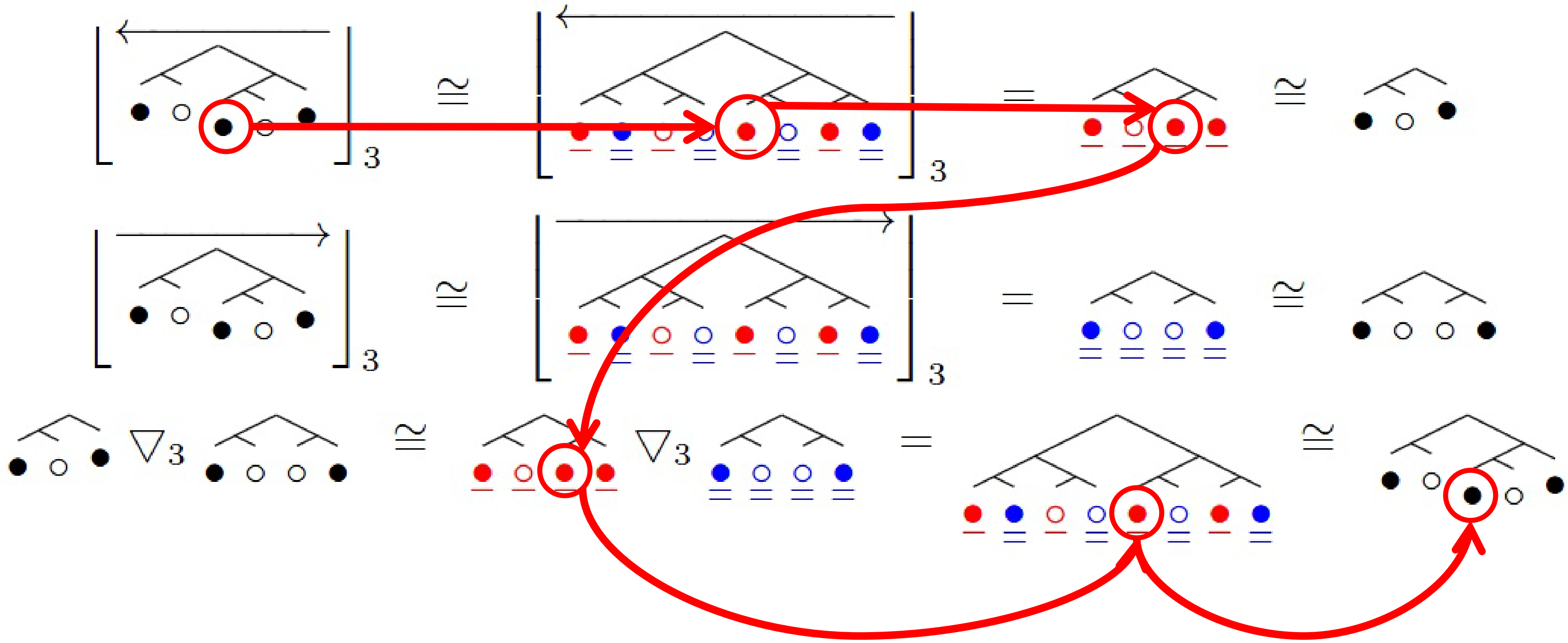


Key properties of averaging

1. Averaging is the inverse of rounding, i.e.,

$$\lfloor \overleftarrow{\tau} \rfloor_n \ \nabla_n \ \lfloor \overrightarrow{\tau} \rfloor_n = \tau$$

- "Proof."

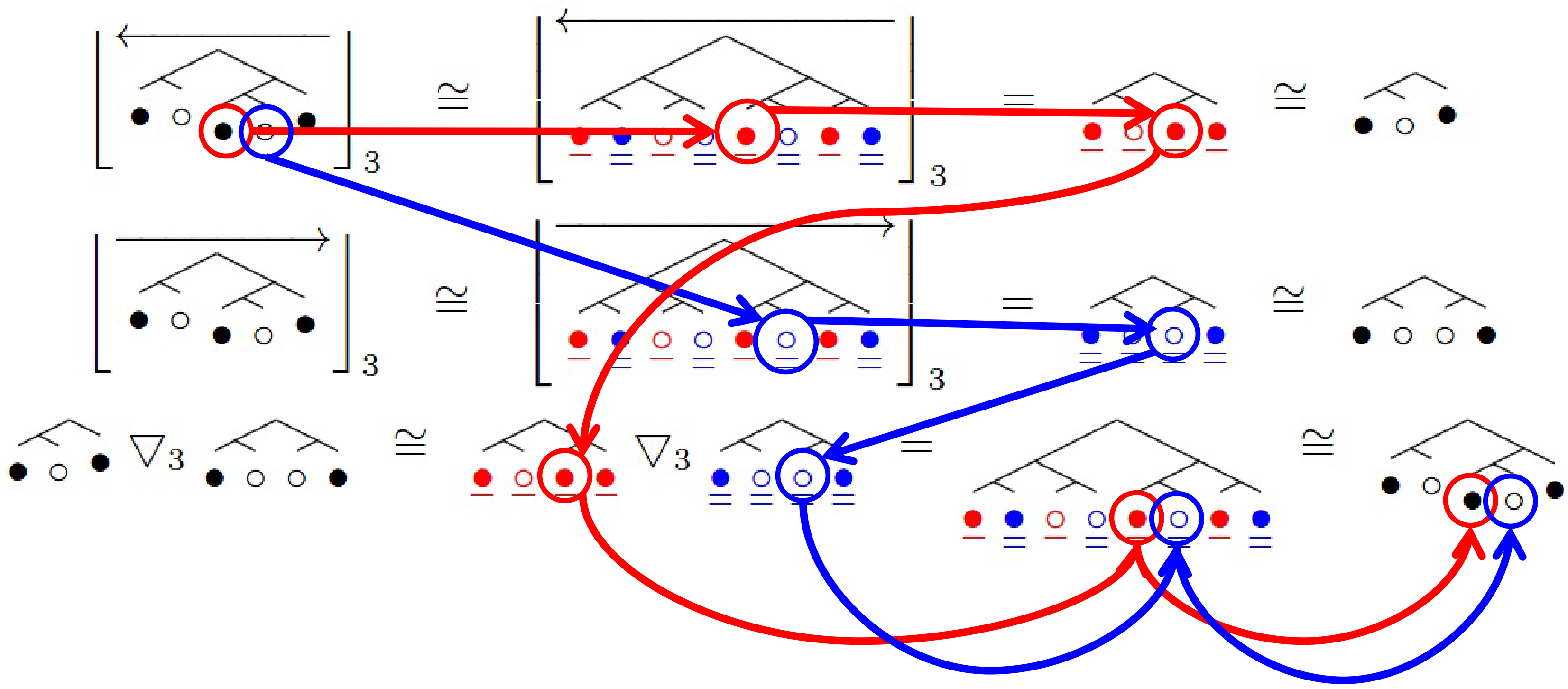


Key properties of averaging

1. Averaging is the inverse of rounding, i.e.,

$$\lfloor \overleftarrow{\tau} \rfloor_n \ \nabla_n \ \lfloor \overrightarrow{\tau} \rfloor_n = \tau$$

- “Proof.”

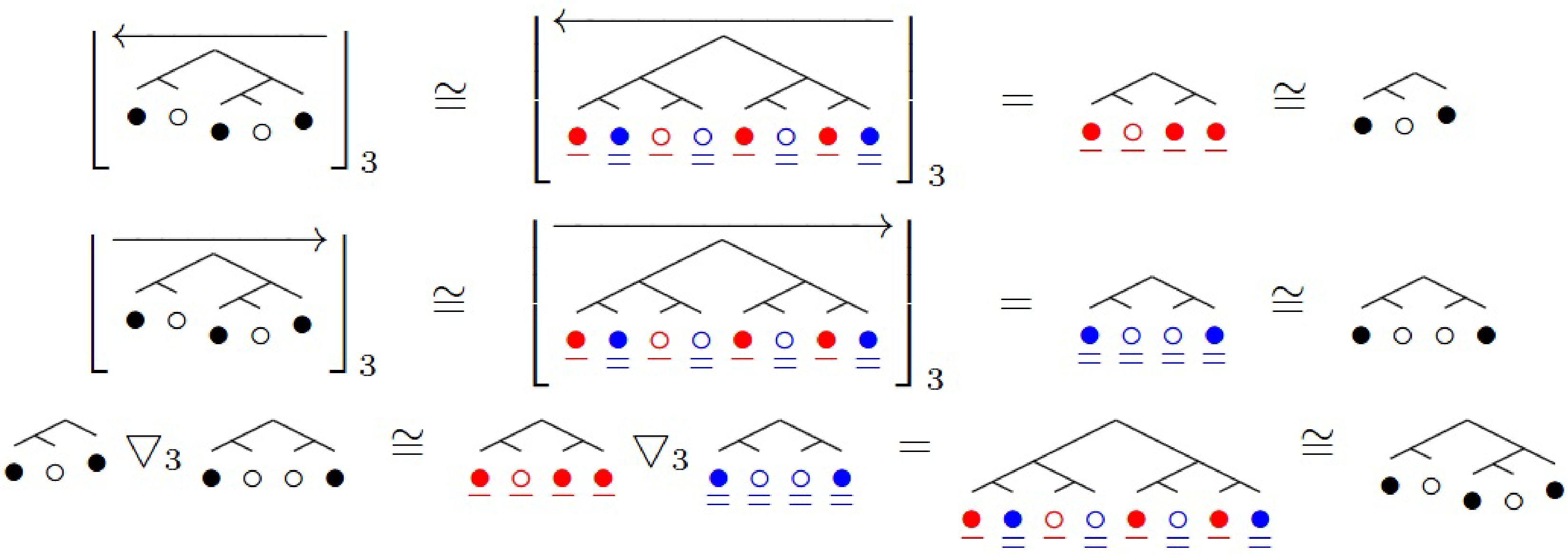


Key properties of averaging

1. Averaging is the inverse of rounding, i.e.,

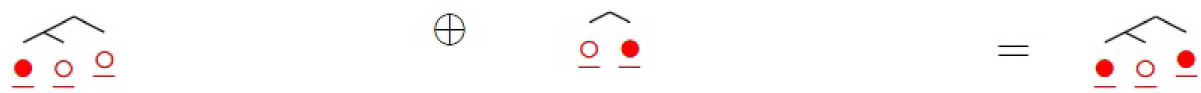
$$\lfloor \overleftarrow{\tau} \rfloor_n \ \nabla_n \ \lfloor \overrightarrow{\tau} \rfloor_n = \tau$$

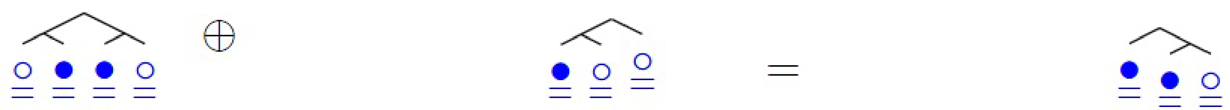
- "Proof."



2. If $\tau_1 \oplus \tau_2 = \tau_3$ and $\tau_1' \oplus \tau_2' = \tau_3'$, then $(\tau_1 \ \nabla_n \ \tau_1') \oplus (\tau_2 \ \nabla_n \ \tau_2') = (\tau_3 \ \nabla_n \ \tau_3')$.

“Proof.”

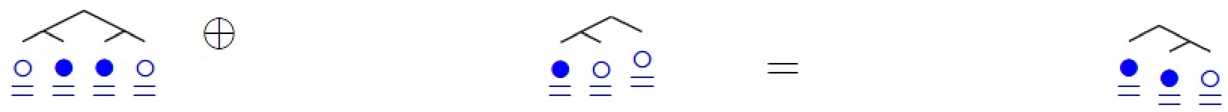
Premise 

Premise 

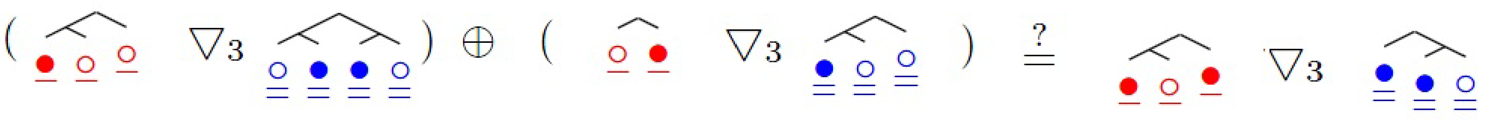
“Proof.”

Premise

Premise



Apply average



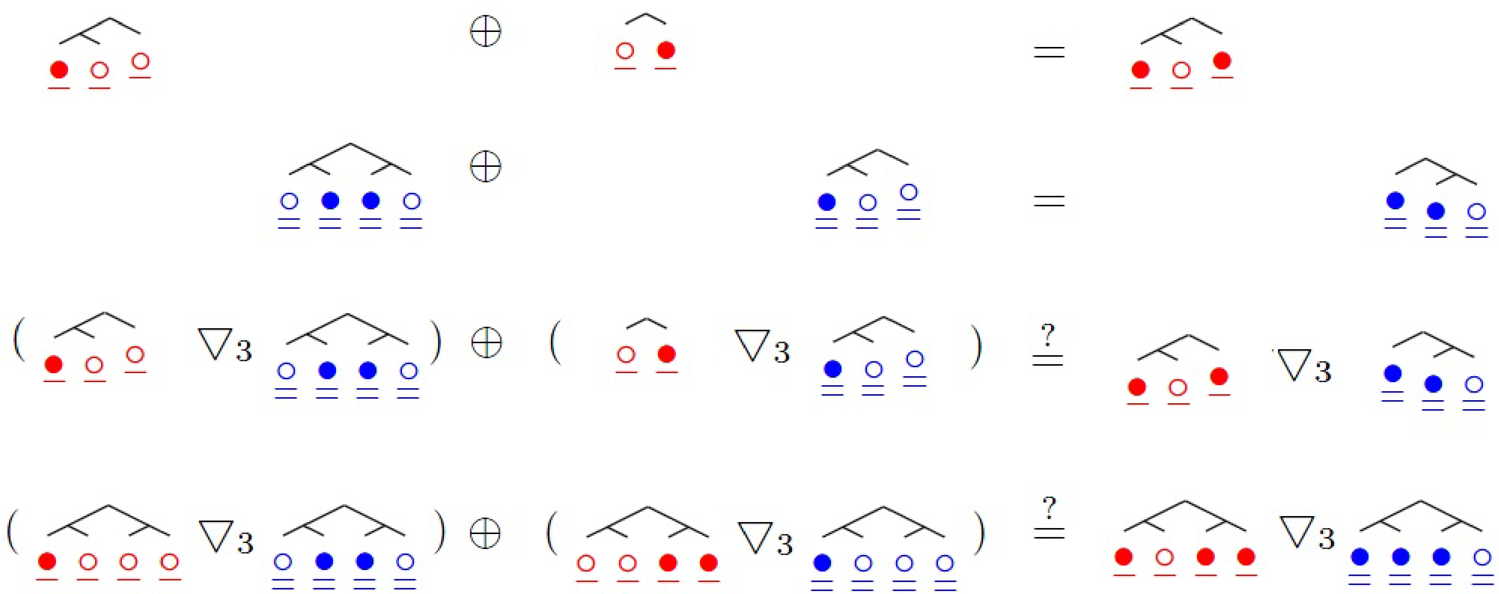
“Proof.”

Premise

Premise

Apply average

Unfold



“Proof.”

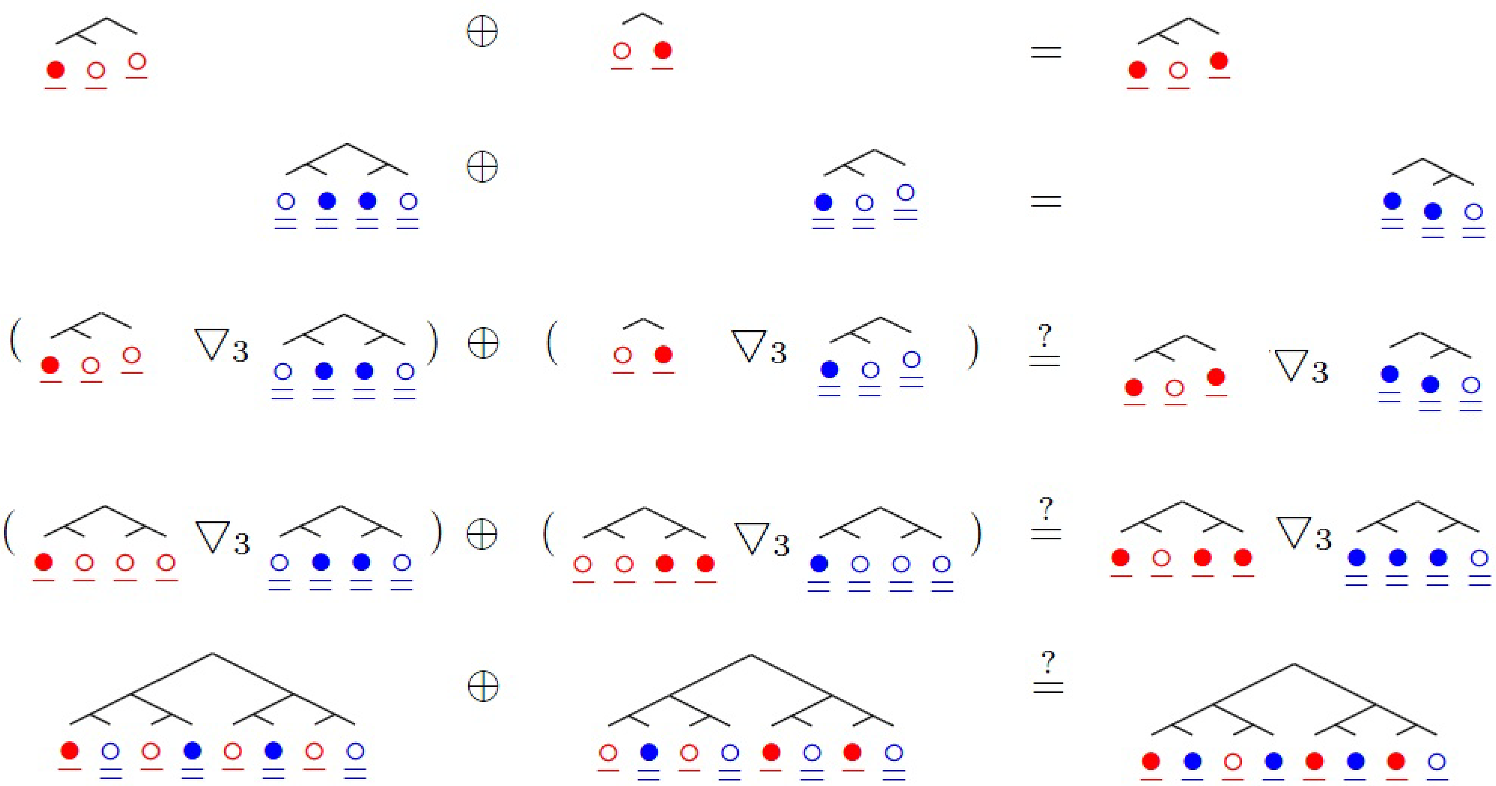
Premise

Premise

Apply average

Unfold

Calculate



“Proof.”

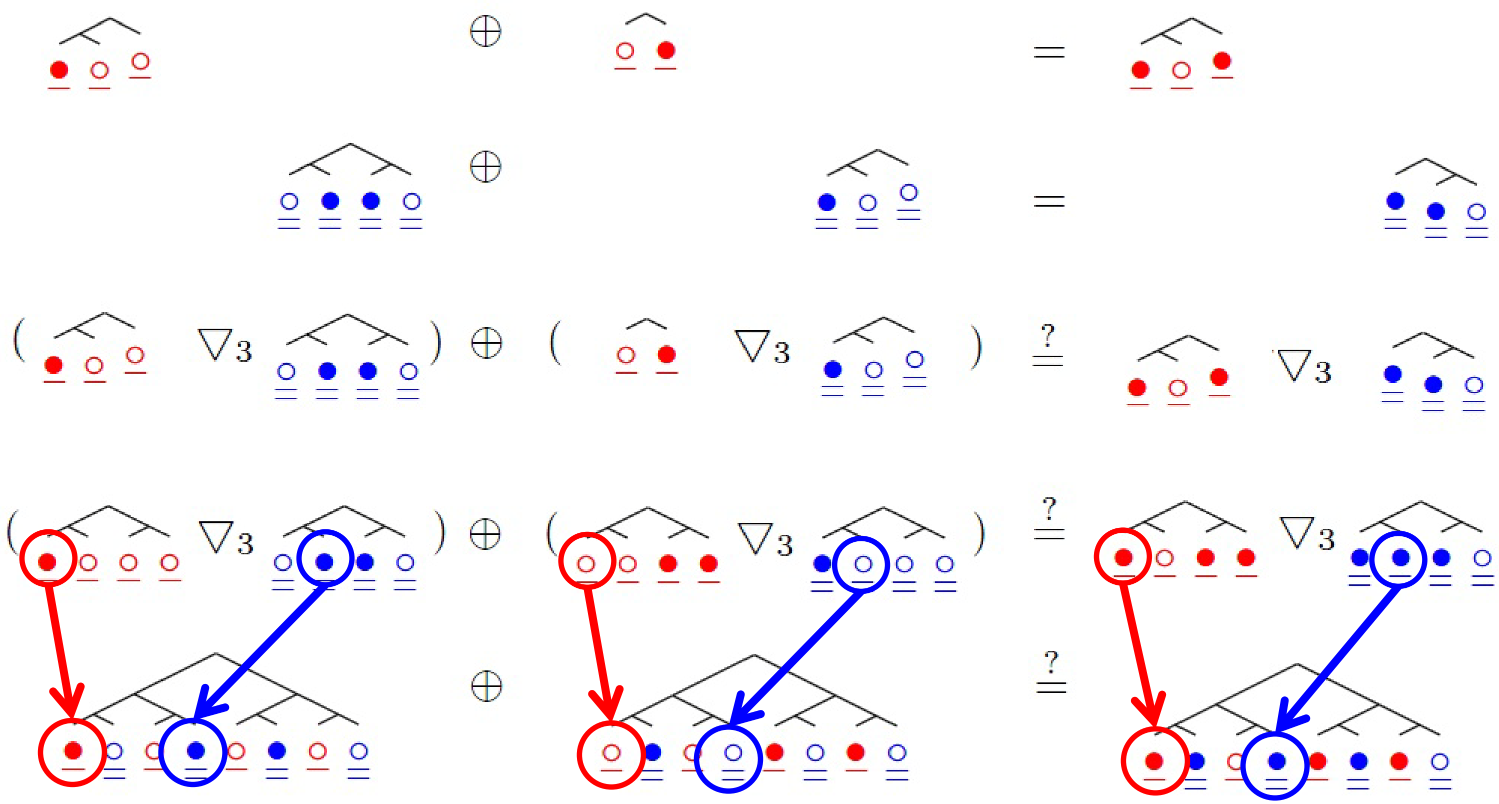
Premise

Premise

Apply average

Unfold

Calculate



Again, because joining occurs leafwise, if they joined before they join after!

“Proof.”

Premise

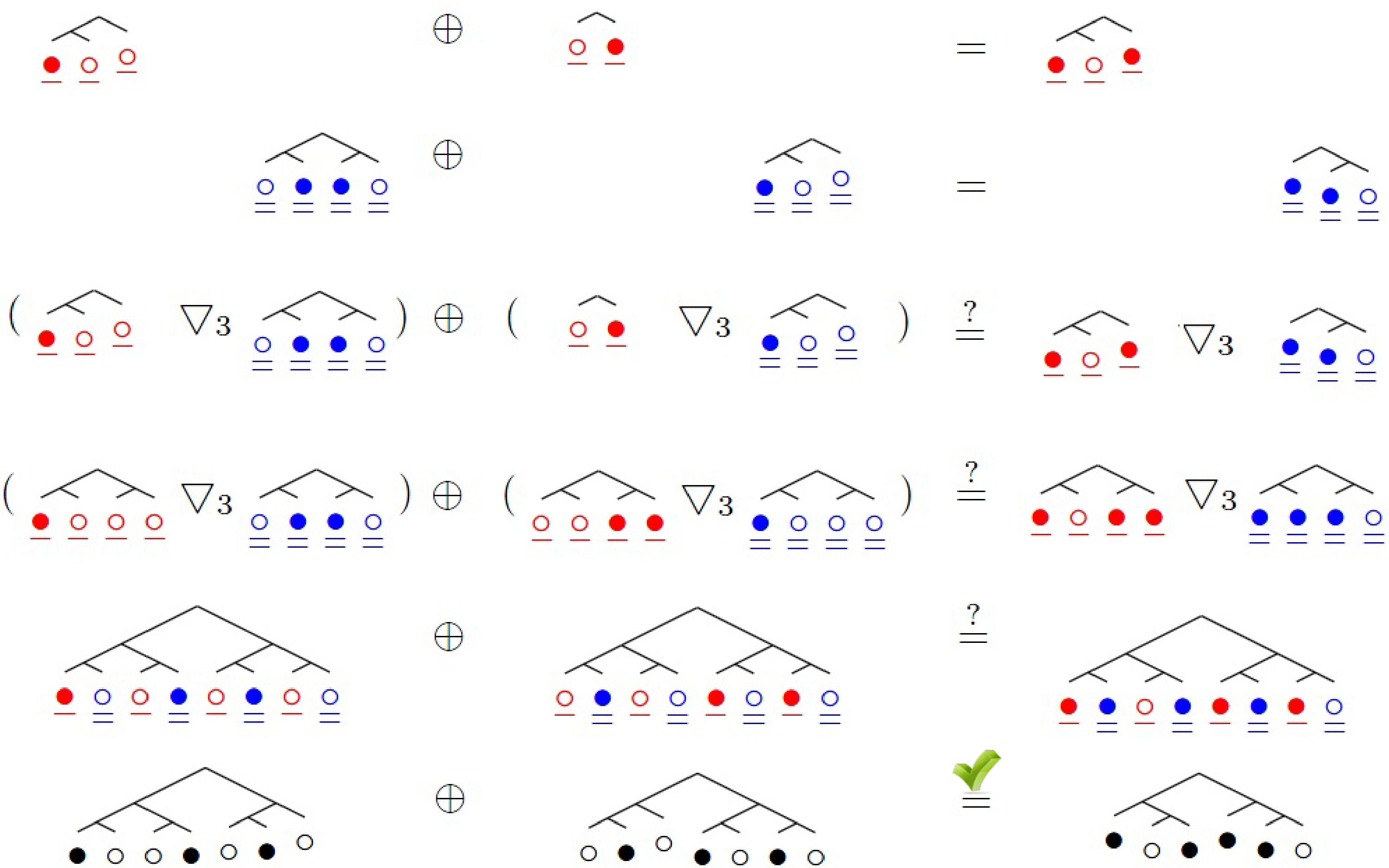
Premise

Apply average

Unfold

Calculate

Refold



Proof sketch: finite IMPL

- Theorem 2: finite search for IMPL
  - Given Σ and Σ', $(\forall\, \sigma.\ \sigma \models \Sigma \rightarrow \sigma \models \Sigma')$ iff
    $(\forall\, \sigma.\ |\sigma|=|\Sigma| \rightarrow \sigma \models \Sigma \rightarrow \sigma \models \Sigma')$
- $\rightarrow$: trivial.
- $\leftarrow$: Consider the case when $|\sigma|=|\Sigma|+1$. By the rounding lemmas, both the left round σ_l and right round σ_r of σ are still solutions for Σ (and have height $|\Sigma|$). Then we apply our hypothesis to learn that σ_l and σ_r are also solutions of Σ'. By averaging property 2, their average is a solution of Σ', and by averaging property 1, their average is equal to σ.

Plan of attack

1. Fractional Shares ✔
2. Verification Tools ✔
3. Our Decision Procedures ✔
4. Completeness ✔
5. Experimental Results

HIP/SLEEK Embedding

	SAT					IMPL				
test	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)
barrier-weak	116	0.4	610	73	530	222	2.1	650	42	450
barrier-strong	116	0.6	660	73	510	222	2.2	788	42	460
barrier-paper	116	0.7	664	73	510	216	2.2	757	42	460
barrier-paper-ex	114	0.8	605	71	520	212	2.3	610	40	430
fractions	63	0.1	0.1	0	0	89	0.1	110	11	110
fractions1	11	0.1	0.1	0	0	15	0.1	31.3	3	30
barrier	68	0.1	0.9	0	0	174	1.2	3.9	0	0
barrier3	36	0.2	0.1	0	0	92	0.2	2.2	0	0
barrier4	59	0.1	0.7	0	0	140	0.9	2.4	0	0
read_ops	14	FAIL	210	14	208	27	FAIL	317	9	150
construct	4	FAIL	70	4	65	17	FAIL	880	17	270
join_ent	3	FAIL	70	3	30	3	FAIL	50	3	48

- Old tool is very fast…

HIP/SLEEK Embedding

	SAT					IMPL				
test	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)
barrier-weak	116	0.4	610	73	530	222	2.1	650	42	450
barrier-strong	116	0.6	660	73	510	222	2.2	788	42	460
barrier-paper	116	0.7	664	73	510	216	2.2	757	42	460
barrier-paper-ex	114	0.8	605	71	520	212	2.3	610	40	430
fractions	63	0.1	0.1	0	0	89	0.1	110	11	110
fractions1	11	0.1	0.1	0	0	15	0.1	31.3	3	30
barrier	68	0.1	0.9	0	0	174	1.2	3.9	0	0
barrier3	36	0.2	0.1	0	0	92	0.2	2.2	0	0
barrier4	59	0.1	0.7	0	0	140	0.9	2.4	0	0
read_ops	14	FAIL	210	14	208	27	FAIL	317	9	150
construct	4	FAIL	70	4	65	17	FAIL	880	17	270
join_ent	3	FAIL	70	3	30	3	FAIL	50	3	48

- But it is incomplete… first two groups of tests were tweaked to avoid the (many) “dark zones”

HIP/SLEEK Embedding

	SAT					IMPL				
test	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)
barrier-weak	116	0.4	610	73	530	222	2.1	650	42	450
barrier-strong	116	0.6	660	73	510	222	2.2	788	42	460
barrier-paper	116	0.7	664	73	510	216	2.2	757	42	460
barrier-paper-ex	114	0.8	605	71	520	212	2.3	610	40	430
fractions	63	0.1	0.1	0	0	89	0.1	110	11	110
fractions1	11	0.1	0.1	0	0	15	0.1	31.3	3	30
barrier	68	0.1	0.9	0	0	174	1.2	3.9	0	0
barrier3	36	0.2	0.1	0	0	92	0.2	2.2	0	0
barrier4	59	0.1	0.7	0	0	140	0.9	2.4	0	0
read_ops	14	FAIL	210	14	208	27	FAIL	317	9	150
construct	4	FAIL	70	4	65	17	FAIL	880	17	270
join_ent	3	FAIL	70	3	30	3	FAIL	50	3	48

- New tool is slower, although **the rest of HIP/SLEEK takes more 3,000ms** on the first four tests

HIP/SLEEK Embedding

	SAT					IMPL				
test	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)
barrier-weak	116	0.4	610	73	530	222	2.1	650	42	450
barrier-strong	116	0.6	660	73	510	222	2.2	788	42	460
barrier-paper	116	0.7	664	73	510	216	2.2	757	42	460
barrier-paper-ex	114	0.8	605	71	520	212	2.3	610	40	430
fractions	63	0.1	0.1	0	0	89	0.1	110	11	110
fractions1	11	0.1	0.1	0	0	15	0.1	31.3	3	30
barrier	68	0.1	0.9	0	0	174	1.2	3.9	0	0
barrier3	36	0.2	0.1	0	0	92	0.2	2.2	0	0
barrier4	59	0.1	0.7	0	0	140	0.9	2.4	0	0
read_ops	14	FAIL	210	14	208	27	FAIL	317	9	150
construct	4	FAIL	70	4	65	17	FAIL	880	17	270
join_ent	3	FAIL	70	3	30	3	FAIL	50	3	48

- Most of the time is spent in the SMT solver (and communication/process overhead)

HIP/SLEEK Embedding

	SAT					IMPL				
test	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)	call no.	BndP (ms)	ShP (ms)	SAT no.	SAT (ms)
barrier-weak	116	0.4	610	73	530	222	2.1	650	42	450
barrier-strong	116	0.6	660	73	510	222	2.2	788	42	460
barrier-paper	116	0.7	664	73	510	216	2.2	757	42	460
barrier-paper-ex	114	0.8	605	71	520	212	2.3	610	40	430
fractions	63	0.1	0.1	0	0	89	0.1	110	11	110
fractions1	11	0.1	0.1	0	0	15	0.1	31.3	3	30
barrier	68	0.1	0.9	0	0	174	1.2	3.9	0	0
barrier3	36	0.2	0.1	0	0	92	0.2	2.2	0	0
barrier4	59	0.1	0.7	0	0	140	0.9	2.4	0	0
read_ops	14	FAIL	210	14	208	27	FAIL	317	9	150
construct	4	FAIL	70	4	65	17	FAIL	880	17	270
join_ent	3	FAIL	70	3	30	3	FAIL	50	3	48

- And, the new procedures are complete!

Standalone

- It’s actually really hard to develop tests to aggressive exercise the share procedures – in lots of code it will happen, but finding small examples is tricky.

Standalone

- It’s actually really hard to develop tests to aggressive exercise the share procedures – in lots of code it will happen, but finding small examples is tricky.

- We developed a standalone benchmark of 53 SAT and 50 IMPLY queries to stress the solver.

Standalone

- It’s actually really hard to develop tests to aggressive exercise the share procedures – in lots of code it will happen, but finding small examples is tricky.

- We developed a standalone benchmark of 53 SAT and 50 IMPLY queries to stress the solver.

- Our new solver solved the entire suite in 1.4s.

Standalone

- It's actually really hard to develop tests to aggressive exercise the share procedures – in lots of code it will happen, but finding small examples is tricky.
- We developed a standalone benchmark of 53 SAT and 50 IMPLY queries to stress the solver.
- Our new solver solved the entire suite in 1.4s.
- **Our old solver could solve fewer than 10%.**